Abstract interpretation-based approaches to Security

A Survey on Abstract Non-Interference and its Challenging Applications

Isabella Mastroeni
Department of Computer Science
University of Verona
Verona, Italy
isabella.mastroeni@univr.it

In this paper we provide a survey on the framework of abstract non-interference. In particular, we describe a general formalization of abstract non-interference by means of three dimensions (observation, protection and semantics) that can be instantiated in order to obtain well known or even new weakened non-interference properties. Then, we show that the notions of abstract non-interference introduced in language-based security are instances of this more general framework which allows to better understand the different components of a non-interference policy. Finally, we consider two challenging research fields concerning security where abstract non-interference seems a promising approach providing new perspectives and new solutions to open problems: Code injection and code obfuscation.

1 Introduction

Understanding information-flow is essential in code debugging, program analysis, program transformation, and software verification but also in code protection and malware detection. Capturing information-flow means modeling the properties of control and data that are transformed dynamically at run-time. Program slicing needs information-flow analysis for separating independent code; code debugging and testing need models of information-flow for understanding error propagation, language-based security needs information-flow analysis for protecting data confidentiality from erroneous or malicious attacks while data are processed by programs. In code protection information-flow methods can be used for deciding where to focus the obfuscating techniques, while in malware detection information flow analyses can be used for understanding how the malware interact with its context or how syntactic metamorphic transformations interfere with the analysis capability of the malware detector. The key aspect in information-flow analysis is understanding the degree of independence of program objects, such as variables and statements. This is precisely captured by the notion of *non-interference* introduced by Goguen and Meseguer [20] in the context of the research on security polices and models.

The standard approach to language-based *non-interference* is based on a characterization of the attacker that does not impose any observational or complexity restriction on the attackers' power. This means that, the attackers have *full power*, namely they are modeled without any limitation in their quest to obtain confidential information. For this reason non-interference, as defined in the literature, is an extremely restrictive policy. The problem of refining this kind of security policy has been addressed by many authors as a major challenge in language-based information-flow security [31]. Refining security policies means weakening standard non-interference checks, in such a way that these restrictions can be used in practice or can reveal more information about how information flows in programs.

In the literature, we can find mainly two different approaches for weakening non-interference: by constraining the power of the attacker (from the observational or the computational point of view), or by allowing some confidential information to flow (the so called *declassification*). There are several works dealing with both these approaches, but to the best of our knowledge, the first approach aiming at characterizing at the same time both the power of the attacker's model and the private information that can flow is *abstract non-interference* [15] where the attacker is modeled as an abstraction of public data (input and output) and the information that may flow is an abstraction of confidential inputs. In this framework these two aspects are related by an adjunction relation [16] formally proving that the more concrete the analysis the attacker can perform the less information we can keep protected.

In this paper, we introduce the abstract non-interference framework from a more general point of view. Data are simply partitioned in unobservable (called *internal*) and *observable* [17] and we may observe also relations between internal and observable data, and not simply attribute independent properties [4]. Our aim is that of showing that the abstract non-interference framework may be exported, due to its generality, to different fields of computer science, providing new perspectives for attacking both well known and new security challenges.

Paper outline. The paper is structured as follows. In the following of this section we introduce the basic notions of abstract interpretation, abstract domain completeness and program semantics, used in the rest of the paper. In Sect. 2 we recall and we slightly generalize the notion of abstract non-interference (`ANI`) formalized in the last years. In particular we describe `ANI` by means of three general dimensions: semantic, observation and protection. Finally we combine all these dimensions together. In Sect. 3 we provide a survey about how, in the literature, this notion of `ANI` has been used for characterizing weakened policies of non-interference in language-based security. Again, we organize the framework by means of the three dimensions that here become: *Who* attacks, *What* is disclosed and *Where/When* the attacker observes. Finally, we conclude the paper in Sect. 4 where we introduce two promising security fields where we believe the `ANI`-based approach may be fruitful for providing a new perspective and a set of new formal tools for reasoning on challenging security-related open problems.

Abstract interpretation: Domains and surroundings. Abstract interpretation is a general theory for specifying and designing approximate semantics of program languages [10]. Approximation can be equivalently formulated either in terms of Galois connections or closure operators [11]. An *upper closure operator* $\rho : C \to C$ on a poset C ($uco(C)$ for short), representing concrete objects, is monotone, idempotent, and extensive: $\forall x \in C.\, x \leq_C \rho(x)$. The upper closure operator is the function that maps the concrete values to their abstract properties, namely with the best possible approximation of the concrete value in the abstract domain. For example, $\mathit{Sign} : \wp(\mathbb{Z}) \to \wp(\mathbb{Z})$, on the powerset of integers, associates each set of integers with its sign: $\mathit{Sign}(\varnothing) = \varnothing \overset{\text{def}}{=}$ *"none"*, $\mathit{Sign}(S) = \{n \mid n > 0\} \overset{\text{def}}{=} +$ if $\forall n \in S.\, n > 0$, $\mathit{Sign}(0) = \{0\} \overset{\text{def}}{=} 0$, $\mathit{Sign}(S) = \{n \mid n < 0\} \overset{\text{def}}{=} -$ if $\forall n \in S.\, n < 0$, $\mathit{Sign}(S) = \{n \mid n \geq 0\} \overset{\text{def}}{=} 0+$ if $\forall n \in S.\, x \geq 0$, $\mathit{Sign}(S) = \{n \mid n \leq 0\} \overset{\text{def}}{=} 0-$ if $\forall n \in S.\, n \leq 0$ and $\mathit{Sign}(S) = \mathbb{Z} \overset{\text{def}}{=}$ *"I don't know"* otherwise. Analogously, the operator $\mathit{Par} : \wp(\mathbb{Z}) \to \wp(\mathbb{Z})$ associates each set of integers with its par-

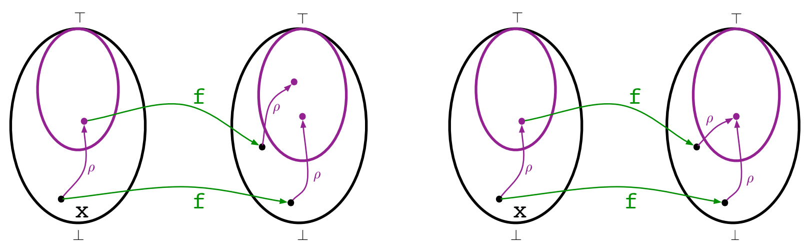

Figure 1: Backward completeness

ity, *Par*$(\varnothing) = \varnothing \overset{\text{def}}{=}$ *"none"*, *Par*$(S) = \{n \in \mathbb{Z} \mid n \text{ is even}\} \overset{\text{def}}{=} \texttt{ev}$ if $\forall n \in S.\, n$ is even, *Par*$(S) = \{n \in \mathbb{Z} \mid n \text{ is odd}\} \overset{\text{def}}{=} \texttt{od}$ if $\forall n \in S.\, n$ is odd and *Par*$(S) = \mathbb{Z} \overset{\text{def}}{=}$ *"I don't know"* otherwise. Usually, *"none"* and *"I don't know"* are simply denoted $\varnothing$ and $\mathbb{Z}$. Formally, closure operators ρ are uniquely determined by the set of their fix-points (or idempotents) $\rho(C)$, for instance *Par* $= \{\mathbb{Z}, \texttt{ev}, \texttt{od}, \varnothing\}$. For upper closures, $X \subseteq C$ is the set of fix-points of $\rho \in uco(C)$ iff X is a *Moore-family* of C, i.e., $X = \mathscr{M}(X) \overset{\text{def}}{=} \{\wedge S \mid S \subseteq X\}$ — where $\wedge\varnothing = \top \in \mathscr{M}(X)$. The set of all upper closure operators on C, denoted $uco(C)$, is isomorphic to the so called *lattice of abstract interpretations of* C [11]. If C is a complete lattice then $uco(C)$ ordered point-wise is also a complete lattice, $\langle uco(C), \sqsubseteq, \sqcup, \sqcap, \top, id\rangle$ where for every $\rho, \eta \in uco(C)$, $I \subseteq \mathsf{Nats}$, $\{\rho_i\}_{i\in I} \subseteq uco(C)$ and $x \in C$: $\rho \sqsubseteq \eta$ iff $\forall y \in C.\, \rho(y) \leq \eta(y)$ iff $\eta(C) \subseteq \rho(C)$; $(\sqcap_{i\in I}\rho_i)(x) = \wedge_{i\in I}\rho_i(x)$; and $(\sqcup_{i\in I}\rho_i)(x) = x \Leftrightarrow \forall i \in I.\, \rho_i(x) = x$.

Abstract interpretation is a theory for approximating program behaviour by approximating their semantics. Now, we formally introduce the notion of precision in terms of abstract domain completeness. There are two kinds of completeness, called *backward* and *forward* completeness [18]. Backward completeness ($\mathscr{B}$) requires accuracy when we compare the computations on the program input domain: the abstract outputs of the concrete computation f are the same abstract outputs obtained by computing the program on the abstract values. Formally, ρ is backward complete for f iff $\rho \circ f \circ \rho = \rho \circ f$ [11]. Consider Fig. 1. The outer oval always represents the concrete domain, while the inner one represents the abstract domain characterised by the closure ρ. The computation is represented by the function f. Hence, on the left, we have incompleteness since the abstract computation on the abstract values ($\rho(f(\rho(x)))$) loses precision with respect to (i.e., is more abstract than) the abstraction of the computation on the concrete values ($\rho(f(x))$). On the right, we have completeness because the two abstract computations coincide. Forward completeness ($\mathscr{F}$) requires accuracy when we compare the abstract and the concrete computations on the output domain of the program, i.e., we compare whether the abstract and the concrete outputs are the same when the program computes on the abstract values. Formally, given a semantics f and a closure ρ, ρ is forward complete for f iff $\rho \circ f \circ \rho = f \circ \rho$. Consider Fig. 2 . On the left, we have incompleteness since the concrete and the abstract computations on abstract values (respectively $f(\rho(x))$ and $\rho(f(\rho(x)))$) does not provide the same result, and in particular the abstraction of the computation loses precision. On the right, the two computations coincide since f returns, as output, an element in ρ, and therefore we have completeness. Finally we observe that, if f is additive[1], then there exists $f^+ \overset{\text{def}}{=} \lambda x. \bigvee\{\, y \mid f(y) \leq x\}$ and we have that ρ is $\mathscr{B}$-complete for f iff it is $\mathscr{F}$-complete for f^+.

[1]It commutes with least upper bounds.

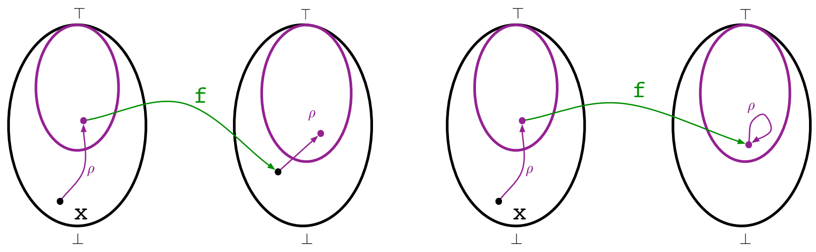

Figure 2: Forward completeness

Programs and Semantics. Consider a simple (deterministic) imperative language $\mathscr{L}$: $\mathtt{C}$ ::= **skip** | $x := e$ | $\mathtt{C}_0$;$\mathtt{C}_1$ | **while** B **do** $\mathtt{C}$ **endw** | **if** B **then** $\mathtt{C}_0$ **else** $\mathtt{C}_1$. Let $\mathbb{P}_{\mathscr{L}}$ be a set of programs in the language $\mathscr{L}$, $\mathit{Var}(P)$ the set of all the variables in $P \in \mathbb{P}_{\mathscr{L}}$, $\mathbb{V}$ be the set of values, $\mathbb{M} : \mathit{Var}(P) \to \mathbb{V}$. In sake of simplicity, in the examples, we consider a fine-grained big-step operational semantics, where each step of computation corresponds to the execution of a single statement, e.g. the execution of an **if** statement corresponds to one step. In this way, the program points coincide with the steps of computation. This is not a mandatory choice and the whole framework can be extended to trace semantics of any granularity. Let $\to_P$ be the transition relation induced by the fine-grained big-step semantics, then we denote by $\langle\!| P |\!\rangle$ the set of execution traces of P obtained by fix-point iteration of the transition relation [9]. We denote by $[\![P]\!] : \mathbb{M} \to \mathbb{M}$ the denotational semantics of P associating with initial states the corresponding final states. In other words $[\![P]\!]$ is the I/O abstraction of the trace semantics $\langle\!| P |\!\rangle$ [9].

2 A general framework for Abstract Non-Interference

In this section, we introduce the notion of abstract non-interference [15], i.e., a weakening of non-interference given by means of abstract interpretations of concrete semantics. We will start from standard notion on non-interference (NI for short), originally introduced in language-based security [7, 20, 31], and here generalized to any kind of classification of data (intended as program variables), where we are interested in understanding whether a given class of data (*internal*) interferes with another class of data (*observable*). In other words, we generalize the public/private data classification in language-based security to a generic observable/internal classification [17].
Consider variables *statically*[2] distinguished into *internal* (denoted $*$) and *observable* (denoted $\circ$). The internal data[3] correspond to those variables that must not interfere with the observable ones. This partition characterises the NI policy we have to verify/define.
Both the input and the output variables are partitioned in this way, and the two partitions need not coincide. Hence, if $\mathscr{I}$ denotes the set of input variables and $\mathscr{O}$ denotes the set of output variables, we have four classes of data: $\mathscr{I}_*$ are the input *internal* variables, $\mathscr{I}_\circ$ are the not internal (potentially observable) input, $\mathscr{O}_*$ are the not observable (hence internal) outputs, and $\mathscr{O}_\circ$ are the output *observables*. Note that the formal distinction between $\mathscr{I}$ and $\mathscr{O}$ is used only to underline that there can be different partitions of

[2]In this paper we do not consider security types that can dynamically change.
[3]In sake of simplicity, we identify data with their containers (variables).

input and output, namely $\mathscr{I}_* = \mathscr{O}_*$ need not hold. In general, we have the same set of variables in input and in output, hence $\mathscr{I} = \mathscr{O} = Var(P)$.
Informally, non-interference can be reformulated by saying that if we fix the values of variables in $\mathscr{I}_\circ$ and we let values of variables in $\mathscr{I}_*$ change, we must not observe any difference in the values of variables in $\mathscr{O}_\circ$. Indeed if this happens it means that $\mathscr{I}_*$ interferes with $\mathscr{O}_\circ$. We will use the following notation: if $n = |\{x \in Var(P) \mid x \text{ is internal}\}| = |\mathscr{I}_*|$, then $\mathbb{I}_* \overset{\text{def}}{=} \mathbb{V}^n$, analogously $\mathbb{O}_* \overset{\text{def}}{=} \mathbb{V}^{|\mathscr{O}_*|}$, $\mathbb{I}_\circ \overset{\text{def}}{=} \mathbb{V}^{|\mathscr{I}_\circ|}$ and $\mathbb{O}_\circ \overset{\text{def}}{=} \mathbb{V}^{|\mathscr{O}_\circ|}$, where $|X|$ denotes the cardinality of the set of variables X. Consider $\mathbb{C} \in \{\mathbb{I}_*, \mathbb{I}_\circ, \mathbb{O}_*, \mathbb{O}_\circ\}$, in the following, we abuse notation by denoting $v \in \mathbb{C}$ the fact that v is a possible tuple of values for the vector of variables evaluated in $\mathbb{C}$, e.g., $v \in \mathbb{I}_*$ is a vector of values for the variables in $\mathscr{I}_*$. Moreover, if x is a tuple of variables in $\mathscr{O}$ (analogous for $\mathscr{I}$) we denote as x^* [resp. x°] the projection of the tuple of variables x only on the variables in $\mathscr{O}_*$ [resp. $\mathscr{O}_\circ$] (analogous for values). At this point, we can reformulate standard non-interference for a deterministic program P[4], w.r.t. fixed partitions of input and output variables $\pi_{\mathtt{I}} \overset{\text{def}}{=} \{\mathscr{I}_\circ, \mathscr{I}_*\}$ and $\pi_{\mathtt{O}} \overset{\text{def}}{=} \{\mathscr{O}_\circ, \mathscr{O}_*\}$ ($\pi = \{\pi_{\mathtt{I}}, \pi_{\mathtt{O}}\}$):

$$\boxed{\begin{array}{c}\text{A program } P\text{, satisfies } \textit{non-interference} \text{ w.r.t. } \pi \text{ if} \\ \forall v \in \mathbb{I}_\circ, \forall v_1, v_2 \in \mathbb{I}_* \,.\, (\llbracket P \rrbracket(v_1, v))^\circ = (\llbracket P \rrbracket(v_2, v))^\circ\end{array}} \tag{1}$$

2.1 An abstract domain completeness problem

In this section, we recall from [16, 4] the completeness formalization of (abstract) non-interference. This characterization underlines that non-interference holds when the abstraction of input and output data (the projection on observable values in standard `NI`) is complete, i.e., precise, w.r.t. the semantics of the program. This exactly means that, starting from a fixed input property the semantics of the program does not change the output observable property [21].
Joshi and Leino's characterization of classic `NI` in [22] provides an equational definition of `NI` which can be easily rewritten as a completeness equation: a program P containing internal and observable variables (ranged over by p and o respectively) satisfies non-interference iff $HH; P; HH = P; HH$, where HH is an assignment of an arbitrary value to p. "The postfix occurrences of HH on each side mean that we are only interested in the final value of o and the prefix HH on the left-hand-side means that the two programs are equal if the final value of o does not depend on the initial value of p" [32].

An abstract interpretation is (backwards) complete for a function, f, if the result obtained when f is applied to any concrete input, x, and the result obtained when f is applied to an abstraction of the concrete input, x, both abstract to the same value. The completeness connection is implicit in Joshi and Leino's definition of secure information flow and the implicit abstraction in their definition is: "each internal value is associated with $\top$, that is, the set of all possible internal values".

Let $\pi \overset{\text{def}}{=} \pi_{\mathtt{I}} = \pi_{\mathtt{O}}$, namely $\mathbb{I}_* = \mathbb{O}_*, \mathbb{I}_\circ = \mathbb{O}_\circ$. The set of program states is $\Sigma = \mathbb{I}_* \times \mathbb{I}_\circ$, which is implicitly indexed by the internal variables followed by the observable variables.

Because HH is an arbitrary assignment to $p \in \mathbb{I}_*$, its semantics can be modelled as an *abstraction function*, $\mathscr{H}$, on sets of concrete program states, Σ; that is, $\mathscr{H} : \wp(\Sigma) \to \wp(\Sigma)$, where $\wp(\Sigma)$ is ordered by subset inclusion, $\subseteq$. For each possible value of an observable variable, $\mathscr{H}$ associates *all* possible values

[4]If P is not deterministic the definition works anyway simply by interpreting $\llbracket P \rrbracket(s)$ as the set of all the possible outputs starting from s.

of the internal variables in P. Thus $\mathscr{H}(X) = \mathbb{I}_* \times X^\circ$, where $\mathbb{I}_*$ is the top element of $\wp(\mathbb{I}_*)$. Hence the Joshi-Leino definition can be rewritten [16] in the following way:

$$\mathscr{H} \circ [\![P]\!] \circ \mathscr{H} = \mathscr{H} \circ [\![P]\!] \tag{2}$$

It is clear that $\mathscr{H}$ is parametric on the partition π (and in general the abstraction $\mathscr{H}$ applied to the input is parametric on $\pi_{\mathtt{I}}$, while the one applied on the output is parametric on $\pi_{\mathtt{O}}$). Anyway, in sake of readability we use simply $\mathscr{H}$ instead of $\mathscr{H}_\pi$.
The equation above is precisely the definition of backwards completeness in abstract interpretation [11, 19] (see [25] for examples). Note that, Equation (2) gives us a way to *dynamically* check whether a program satisfies a confidentiality policy, this is due to the use of denotational semantics. In [25] we show that we can perform the same analysis statically, by involving the weakest precondition semantics.

In particular, static checking involves $\mathscr{F}$-completeness, instead of $\mathscr{B}$-completeness, and the use of weakest preconditions instead of the denotational semantics [25]. With weakest preconditions, (written *Wlp*$_P$), equation (2) has the following equivalent reformulation:

$$\mathscr{H} \circ \mathit{Wlp}_P \circ \mathscr{H} = \mathit{Wlp}_P \circ \mathscr{H} \tag{3}$$

Equation (3) says that $\mathscr{H}$ is $\mathscr{F}$-complete for Wlp_P. The equation asserts that $\mathit{Wlp}_P(\mathscr{H}(X))$ is a fixpoint of $\mathscr{H}$, meaning that $\mathit{Wlp}_P \circ \mathscr{H}$ associates each observable output with any possible internal input: a further abstraction of the fixpoint (cf., the lhs of equation (3)) yields nothing new. Because no *distinctions among internal inputs* get exposed to an observer, the observable output is independent of the internal input, hence also equation (3) asserts classic `NI`.

2.2 Tuning Non-Interference: Three dimensions of Non-Interference

We believe that the real added value of abstract non-interference is the possibility of deeply understanding which are the *actors* playing and which is their role in the definition of a security policy to enforce [25]. In particular, we can observe that in the abstract non-interference framework we can identify three *dimensions*: (a) the semantic dimension, (b) the observation dimension and (c) the protection/declassification dimension. These dimensions are pictorially represented in Fig. 3 [4]. In general, to describe

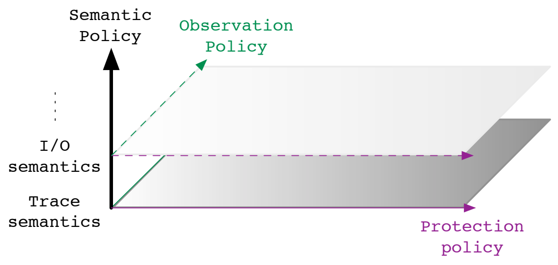

Figure 3: The three dimensions of non-interference

any non-interference property, we first fix its semantic dimension. The semantic dimension comprises of the *concrete semantics* of the program (the solid arrow in Fig. 3 shows Cousot's hierarchy of semantics

[9]), the *set of observation points*, that is the program points where the observer can analyse data, and the *set of protection points*, that is the program points where information must be protected. Next we fix the program's observation and protection dimensions that say *what* information can be observed and *what* information needs to be protected at these program points. This is how the *what* dimension of declassification policies [33] is interpreted in [25].
Consider, for instance standard `NI` in Eq. 1. The three dimensions of `NI` are as follows. The semantic dimension, i.e., the concrete semantics, is P's denotational semantics. Both inputs and outputs constitute the observation points while inputs constitute the protection points because it is internal data at program inputs that need to be protected. The observation dimension is the identity on the observable input-output because the observer can only analyse observable inputs and outputs. Such an observer is the most powerful one for the chosen concrete semantics because its knowledge (that is, the observable projections of the initial and final states of the two runs of the program) is completely given by the concrete semantics. Finally, the protection dimension is the identity on the *internal input*. This means that *all* internal inputs must be protected, or dually, *no* internal inputs must be released.
In this survey we generalize these dimensions in the notion of abstract non-interference introduced in [17], which goes beyond the standard low and high data classification in language-based security.

2.2.1 The semantic dimension

In this section our goal is to provide a general description of a semantic dimension that is parametric on any set of protection and observation points. For this purpose it is natural to move to a trace semantics. The first step in this direction is to consider a function *post* as defined below. Let us define $post_P \overset{\text{def}}{=} \{\langle s,t\rangle \mid s,t\in\Sigma,\ s\rightarrow_P t\}$, where, we recall that $\rightarrow_P$ is the semantic transition relation. From this definition of $post_P$ it is quite straightforward to recover non-interference based on I/O observation [4]. Let $post_P^+$ the transitive closure of $post_P$ associating with each initial state s the final state t reachable from s, i.e., $post_P^+ \overset{\text{def}}{=} \{\langle s,t\rangle \mid s,t\in\Sigma,\ s\rightarrow_P^* t,\ t \text{ final}\}$. Then, an equivalent characterization of `NI`, where $\Sigma_\vdash$ is the set of initial states, is

$$\forall s_1,s_2\in\Sigma_\vdash.\ s_1^\circ = s_2^\circ \ \Rightarrow\ post_P^+(s_1)^\circ = post_P^+(s_2)^\circ$$

`NI` ***for trace semantics.*** A denotational semantics does not take into account the whole history of computation, and thus restricts the kind of protection/declassification policies one can model. In order to handle more precise policies, that take into account *where/when* [33] information is released in addition to *what* information is released we must consider a more concrete semantics, such as trace semantics. More precisely, depending on how we fix the observation points we describe a *where* or a *when* dimension: If the observation points are program points then we are fixing *where* to observe, if the they are computational steps then we are fixing *when* to observe. In our semantics these points coincide and we choose to call this dimension *where*.
Let us define `NI` on traces:

$$\forall s_1,s_2\in\Sigma_\vdash.\ s_1^\circ = s_2^\circ \ \Rightarrow\ \langle\!|P|\!\rangle(s_1)^\circ = \langle\!|P|\!\rangle(s_2)^\circ$$

This definition says that given two observably indistinguishable input states, s_1 and s_2, the two executions of P must generate two — both finite or both infinite — sequences of states in which the corresponding states in each sequence are observably indistinguishable. Equivalently, we can use a set of *post* relations: for $i \in \mathsf{Nats}$ we define the family of relations $post^i_P \overset{\text{def}}{=} \{\langle s,t\rangle \mid t \in \Sigma,\ s \in \Sigma_\vdash,\ s \rightarrow^i_P t\}$, i.e., $post^i_P$ is the I/O semantics after i steps of computations. The following result is straightforward.

Proposition 2.1 *[4]* $\mathtt{NI}$ *on traces w.r.t.* $\pi_\mathtt{I}$ *and* $\pi_\mathtt{O}$ *holds iff for each program point,* i*, of program* P*, we have* $\forall s_1, s_2 \in \Sigma_\vdash . s_1^\circ = s_2^\circ \;\Rightarrow\; post^i_P(s_1)^\circ = post^i_P(s_2)^\circ$.

The *post* characterization of trace-based $\mathtt{NI}$ precisely identifies the observation points as the outputs of the *post* relations, that is, any possible intermediate state of computation, and the protection points are identified as the inputs of the *post* relations, that is, the initial states.

General semantic policies. In the previous paragraph, we show how for denotational and trace semantics, we can define a corresponding set of *post* relations fixing protection and observation points. In order to understand how we can generalize this definition, let us consider a graphical representation of the situations considered in Fig. 4 [25].

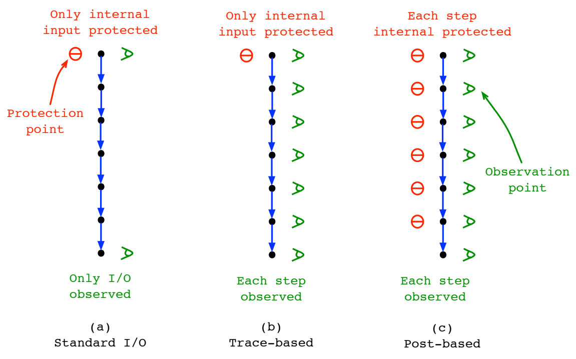

Figure 4: Different notions of non-interference for trace semantics

(i) In the first picture, the semantic dimension says that an observer can analyse the observable inputs and outputs only, while we can protect only the internal inputs. This notion corresponds to $\mathtt{NI}$.

(ii) In the second picture, the semantic dimension says that an observer can analyse each intermediate state of computation, including the input and the output, while the protection point is again the input.

(iii) In the last picture, the semantic dimension says that an observer can analyse each intermediate state of computation, while the protection points are all intermediate states of the computation. In order to check this notion of non-interference for a program we have to check non-interference separately for each statement of the program itself. It is worth noting that this corresponds exactly to $\forall s_1, s_2 \in \Sigma.\ s_1^\circ = s_2^\circ.\ post_P(s_1)^\circ = post_P(s_2)^\circ$.

It is clear that between (i) and (ii) there are several notions of non-interference depending on the intermediate observable states fixed by the where dimension of the policy (*observation points* in Fig. 4). For example, in language-based security, gradual release [2] considers as observation points only those program points corresponding to observable events (i.e., assignment to observable variables, declassification points and termination points). However, unless we consider interactive systems that can provide inputs in arbitrary states, we will protect only the initial ones. Hence, in this case, (ii) and (iii) collapse to the same notion.

Definition 2.2 (Trace-based $\mathtt{NI}$) *[4] Given a set $\mathtt{O}$ of observation points, the notion of $\mathtt{NI}$ w.r.t. $\pi_{\mathtt{I}}$ and $\pi_{\mathtt{O}}$, based on the semantic dimension w.r.t. $\mathtt{O}$, and the input as protection point, is defined:*

$$\forall j \in \mathtt{O}.\, \forall s_1, s_2 \in \Sigma_{\vdash}.\; s_1^{\circ} = s_2^{\circ} \;\Rightarrow\; post_P^j(s_1)^{\circ} = post_P^j(s_2)^{\circ}$$

Note that, this is a general notion, that can be formulated depending on $\mathtt{O}$. In particular, we obtain standard $\mathtt{NI}$ by fixing $\mathtt{O} = \{p \mid p$ is the final program point$\}$, we obtain the most concrete trace-based $\mathtt{NI}$ policy by fixing $\mathtt{O} = \{p \mid p$ is any program point$\}$. But we can also obtain intermediate notions depending on $\mathtt{O}$: we obtain gradual release by fixing $\mathtt{O} = \{p \mid p$ is a program point corresponding to an observable event$\}$.

Example 2.3 *Let us consider the program fragment P with the semantic policies represented in the following picture. The semantic dimension represented in picture (a) is the I/O one. In this case, for each pair of initial states s_1, s_2 such that $s_1^{\circ} = s_2^{\circ}$ we have to check $post^6(s_1)^{\circ} = post^6(s_2)^{\circ}$, and it is clear that this hold since, for instance, both l_1 and l_2 become constant.*

$$\begin{array}{lll}
p_0 \rightarrow & & \leftarrow o_0 \\
 & h_1 := h_2; & \\
 & h_2 := h_2 \bmod 2; & \\
 & l_1 := h_2 & \\
 & l_2 := h_1 & \\
 & l_2 := 5 & \\
 & l_1 := l_2 + 3 & \\
 & & \leftarrow o_6 \\
 & (a) &
\end{array}
\qquad
\begin{array}{lll}
p_0 \rightarrow & & \leftarrow o_0 \\
 & h_1 := h_2; & \\
 & h_2 := h_2 \bmod 2; & \\
 & l_1 := h_2 & \\
 & & \leftarrow o_3 \\
 & l_2 := h_1 & \\
 & & \leftarrow o_4 \\
 & l_2 := 5 & \\
 & l_1 := l_2 + 3 & \\
 & & \leftarrow o_6 \\
 & (b) &
\end{array}$$

On the other hand, the policy in picture (b) considers $\mathtt{O} = \{3,4,6\}$ (the observable events). In this case, for each pair of initial states s_1, s_2 such that $s_1^{\circ} = s_2^{\circ}$ we have to check $post^6(s_1)^{\circ} = post^6(s_2)^{\circ}$, but also $post^4(s_1)^{\circ} = post^4(s_2)^{\circ}$ and $post^3(s_1)^{\circ} = post^3(s_2)^{\circ}$, and both these new tests fail since in all the corresponding program points there is a leakage of private information. For instance, if $s_1 = \langle h_1, h_2 = 2, l_1, l_2\rangle$ and $s_2 = \langle h_1, h_2 = 3, l_1, l_2\rangle$ then we have different l_1 values in o_3: respectively $l_1 = 0$ and $l_1 = 1$.

Semantic dimension in the completeness formalization. We can observe that the completeness equation is parametric on the semantic function used. Since the denotational semantics is the *post* of a transition system where all traces are two-states long. A generalization of the completeness reformulation, in order to cope also with trace-based $\mathtt{NI}$ (Def. 2.2), can be immediately obtained [4]. Theorem 2.4 below shows the connection, also for traces, between completeness and non-interference.

Theorem 2.4 *[4] Let $\langle \Sigma, \rightarrow_P\rangle$ be the transition system for a program P, and $\mathtt{O}$ a set of observation points. Then, trace-based $\mathtt{NI}$ w.r.t. $\pi_{\mathtt{I}}$ and $\pi_{\mathtt{O}}$ (Def. 2.2) holds iff $\forall j \in \mathtt{O}.\, \mathscr{H} \circ \mathrm{post}_P^j \circ \mathscr{H} = \mathscr{H} \circ \mathrm{post}_P^j$.*

This theorem characterizes NI as a family of completeness problems also when a malicious attacker can potentially observe the whole trace semantics, namely when we deal with trace-based NI.

Moreover, let us denote as $\widetilde{pre}^j$ the adjoint map of $post^j$[5] in the same transition system, then the completeness equation can be rewritten as $\mathscr{H} \circ \widetilde{pre}^j \circ \mathscr{H} = \widetilde{pre}^j \circ \mathscr{H}$.

2.2.2 The observation dimension

Consider the program $P \stackrel{\text{def}}{=} x := |x| * Sign(y)$, where $\mathscr{I}_* = \{y\}$ and $\mathscr{I}_\circ = \mathscr{O}_\circ = \{x\}$, suppose that $|\cdot|$ is the absolute value function, then "*only a portion of x is affected, in this case x's sign. Imagine if an observer could only observe x's absolute value and not x's sign*" [7] then we could say that in the program there is non-interference between $*$ and $\circ$. Abstract interpretation provides the most appropriate framework to further develop Cohen's intuition. The basic idea is that an observer can analyze only some properties, modeled as abstract interpretations of the concrete program semantics.

Suppose the observation points fixed by the chosen semantics are input and output. Then the observation dimension might require that the observer analyse a particular *property*, ρ, of the observable output — e.g., parity — and a particular property, η, of the observable input — e.g., signs. In the following, we will consider $\eta \in uco(\mathbb{I})$ such that η abstracts internal and observable variables. This abstraction may be attribute independent or relational[6]. Attribute independent means that η can be split in two independent abstractions identifying precisely what is observable ($\mathbb{I}_\circ$) or not ($\mathbb{I}_*$). Hence, in this case it can be split in one abstraction for the variables in $\mathscr{I}_*$, denoted η_*, and one for the variables in $\mathscr{I}_\circ$, denoted $\eta_\circ$, and we write $\eta = \langle \eta_*, \eta_\circ \rangle$. Then we obtain a weakening of standard NI as follows:

$$\forall x_1, x_2 \in \mathbb{V} \,.\, x_1^\circ = x_2^\circ \;\Rightarrow\; \rho([\![P]\!](\eta(x_1))) = \rho([\![P]\!](\eta(x_2))) \tag{4}$$

This weakening, here called *abstract non-interference (*ANI*)*, was first partially introduced in [15]. The interesting cases are two. The first is $\eta = id$, which consists in a *dynamic* analysis where the observers collects the (possibly huge) set of possible computations and extract in some way (e.g. data mining) properties of interest. The second is an observer performing a *static* analysis of the code, in which case usually $\eta = \rho$. Standard NI is recovered by setting η to be the identity and ρ to the projection on observable values, i.e., $\rho = \langle \mathbb{T}^*, id^\circ \rangle$, where $\mathbb{T}^* \stackrel{\text{def}}{=} \lambda x \in \mathbb{I}_* . \top$ (in this case $\top = \mathbb{I}_*$) and $id^\circ = \lambda x \in \mathbb{I}_\circ . x$ is the identity on observables.

Observation dimension for a generic semantic dimension. In [25] we showed how to combine a semantic dimension that comprises of a trace-based semantics with an observation dimension. In other words, we showed how we abstract a trace by a state abstraction. Consider the following concrete trace where each state in the trace is represented as a pair $\langle x^*, x^\circ \rangle$.

$$\langle 3,1 \rangle \to \langle 2,2 \rangle \to \langle 1,3 \rangle \to \langle 0,4 \rangle \to \langle 0,4 \rangle$$

Suppose also that the trace semantics fixes the observation points to be each intermediate state of computation. Now suppose the observation dimension is that only the parity (represented by the abstract

[5]By adjoint relation the function $\widetilde{pre}^j$ is the weakest precondition of the corresponding function $post^j$

[6]Here, by relational, we mean not attribute independent, namely a property describing relations of elements, for example $\eta(\langle x, y \rangle) = 0+$ if $x + y \geq 0$ is a relational property.

domain, *Par*) of the public data can be observed. Then the observation of the above trace through *Par* is:

$$\langle 3, odd\rangle \to \langle 2, even\rangle \to \langle 1, odd\rangle \to \langle 0, even\rangle \to \langle 0, even\rangle$$

The abstract notion of `NI` on traces is formulated by saying that all the execution traces of a program starting from states with the same property (η) of public input, have to provide the same property (ρ) of reachable states. Therefore, the general notion of `ANI` consists simply in abstracting each state of the computational trace. We thus have

Definition 2.5 (Trace-based `ANI`) *Given a set of observation points* $\mathtt{O}$ *and the partitions* $\pi_{\mathtt{I}}$ *and* $\pi_{\mathtt{O}}$

$$\forall j \in \mathtt{O}. \forall s_1, s_2 \in \Sigma_{\vdash} .\ s_1^\circ = s_2^\circ \ \Rightarrow\ \rho(post_P^j(\eta(s_1))) = \rho(post_P^j(\eta(s_2)))$$

The following example shows the meaning of the observation dimension.

Example 2.6 *Consider the program fragment in Ex. 2.3 together with the semantic policies shown so far. If we consider the semantic dimension in* (a) *and an observer able only to analyse in output the sign of integer variables, i.e.,* $\eta = id$ *and* $\rho = \langle \mathbb{T}^*, \rho_\circ\rangle$, $\rho_\circ = \{\varnothing, <0, \geq 0, \top\}$, *trivially we have that, for any pair of initial states* s_1 *and* s_2 *agreeing on the observable part,* $\rho(post^6(s_1)) = (\geq 0) = \rho(post^6(s_2))$. *Namely, the program is secure. Consider now the semantic dimension in* (b) *and the same observation dimension. In this case, non-interference is still satisfied in* o_6 *but it fails in* o_3 *and in* o_4 *since by changing the sign of* h_2 *we change the sign of respectively* l_1 *and* l_2.

Observation dimension in the completeness formalization. In order to model `ANI` by using the completeness equation we have to embed the abstractions characterizing the attacker into the abstract domain $\mathscr{H}$. Hence, let us first note that $\mathscr{H} = \lambda X . \langle \mathbb{T}^*(X^*), id^\circ(X^\circ)\rangle = \lambda X . \langle \mathbb{I}_*, X^\circ\rangle$ where $X^* \stackrel{\text{def}}{=} \{x^* \mid x \in X\}$, i.e., $\mathscr{H}$ is the product of respectively the top and the bottom abstractions in the lattice of abstract interpretations. As far as the semantics is concerned, if the attacker may perform a static analysis w.r.t. η in input and ρ in output, then the semantics is its best correct approximation, i.e.,

$$[\![P]\!]_\eta^\rho \stackrel{\text{def}}{=} \lambda x . \rho \circ [\![P]\!] \circ \eta(x)$$

This means that the right formalization of `ANI` via completeness is

$$\mathscr{H} \circ [\![P]\!]_\eta^\rho \circ \mathscr{H} = \mathscr{H} \circ [\![P]\!]_\eta^\rho \tag{5}$$

The next theorem shows that the equation above completely characterizes `ANI` as a completeness problem. This theorem is a generalization of the one proved for language-based security [16].

Theorem 2.7 *Consider* $\rho \in uco(\wp(\mathbb{I}))$ *defining what is observable,* $\eta \in uco(\wp(\mathbb{I}))$*:*

$$P \textit{ satisfies } \mathtt{ANI} \textit{ in Eq. 4} \ \Leftrightarrow\ \mathscr{H} \circ [\![P]\!]_\eta^\rho \circ \mathscr{H} = \mathscr{H} \circ [\![P]\!]_\eta^\rho .$$

If $\eta = id$, and $\rho = \langle \rho_*, \rho_\circ\rangle \in uco(\wp(\mathbb{I}))$, we define $\mathscr{H}_\rho \in uco(\wp(\mathbb{I}))$: $\mathscr{H}_\rho \stackrel{\text{def}}{=} \lambda X . \mathscr{H}(X) \circ \rho = \langle \mathbb{I}_*, \rho_\circ(X^\circ)\rangle \in uco(\wp(\mathbb{I}))$. In this case we can rewrite the completeness characterization as

$$\mathscr{H}_\rho \circ [\![P]\!] \circ \mathscr{H} = \mathscr{H}_\rho \circ [\![P]\!] \tag{6}$$

2.2.3 The protection dimension

Suppose now, we aim at describing a property on the input representing for which inputs, we are interested in testing NI properties [4]. Let $\phi \in uco(\wp(\mathbb{I}))$ such a property, also this input property can be modeled, when possible, as an attribute independent abstraction of states, i.e., $\phi = \langle \phi_*, \phi_\circ \rangle$. This component of the NI dimension specifies *what* must be protected or dually, what must be declassified.
For denotational semantics, standard NI says that nothing must be declassified. Formally, we can say that the property $\mathbb{T}^*$ has been declassified. From the observer perspective, this means that every internal input has been mapped to $\top$. On the other hand, suppose we want to declassify the parity of the internal inputs. Then we do not care if the observer can analyse any change due to the variation of parity of internal inputs. For this reason, we only check the variations of the output when the internal inputs have the same parity property. Formally, consider an abstract domain ϕ — the selector, that is a function that maps any internal input to its corresponding parity [7]. Then a program P satisfies declassified non-interference (DNI) provided

$$\forall x_1, x_2 \in \mathbb{V} \,.\, \phi(x_1) = \phi(x_2) \;\Rightarrow\; [\![P]\!](x_1)^\circ = [\![P]\!](x_2)^\circ \qquad (7)$$

This notion of DNI has been advanced several times in the literature, e.g., by [30]; this particular formulation using abstract interpretation where $\phi \overset{\text{def}}{=} \langle \phi_*, id^\circ \rangle$ is due to [15] and is further explained in [24] where it is termed "declassification by allowing". The generalization of DNI to Def. 2.2 is straightforward. Since we protect the internal inputs, we simply add the condition $\phi(s_1) = \phi(s_2)$ to Def. 2.2. In general, we can consider a different declassification policy ϕ_j for each observation point o_j, $j \in \mathtt{O}$.
Definition 2.8 (Trace-based DNI.) *Let* $\mathtt{O}$ *be a set of observation points,* $\pi_{\mathtt{I}}$ *a partition of inputs and* $\pi_{\mathtt{O}}$ *the partitions of observable outputs. Let* $\forall j \in \mathtt{O}$ ϕ_j *be the input property declassified in* o_j : $\forall j \in \mathtt{O}. \forall s_1, s_2 \in \Sigma_{\vdash} \,.\, \phi_j(s_1) = \phi_j(s_2) \Rightarrow\; post^j_P(s_1)^\circ = post^j_P(s_2)^\circ$
This definition allows a versatile interpretation of the relation between the *where* and the *what* dimensions for declassification [25]. Indeed if we have a unique declassification policy, ϕ, that holds for all the observation points, then it means that $\forall j \in \mathtt{O}. \phi_j = \phi$. On the other hand, if we have explicit declassification, then we have two different choices. We can combine *where* and *what* supposing that the attacker knowledge can only increase. We call this kind of declassification *incremental* declassification. On the other hand, we can combine *where* and *what* in a stricter way. Namely, the information is declassified only in the particular observation point where it is explicitly declared, but not in the following points. In this case it is sufficient to consider as ϕ_j exactly the property corresponding to the explicit declassification and we call it *localized* declassification. This kind of declassification can be useful when we consider the case where the information obtained by the different observation points cannot be combined, for example if different points are observed by different observers, as it can happen in the security protocol context.

Protection dimension in the completeness formalization. Finally, we can model the protection dimension as a completeness problem. Consider an abstract domain $\phi = \langle \phi_*, \phi_\circ \rangle \in uco(\wp(\mathbb{I}))$, where $\phi_* \in uco(\wp(\mathbb{I}_*))$ and $\phi_\circ \in uco(\wp(\mathbb{I}_\circ))$. In this case we consider the concrete semantics $[\![P]\!]$ of the program, since the property ϕ is only used for deciding/selecting when we have to check whether non-interference is satisfied or not. For this reason, the property ϕ is embedded in the input abstract domain

[7] More precisely, ϕ maps a set of internal inputs to the join of the parities obtained by applying ϕ to each internal input; the join of $even$ and odd is $\top$.

in a way similar to what we have done for the output observation [25].
Consider any $o \in X^\circ$. Define the set $H_o \overset{\text{def}}{=} \{r \in \mathbb{I}_* \mid \langle r, o\rangle \in X\}$; i.e., given a value o, H_o contains all the relevant values associated with o in X. Then the selecting abstract domain, $\mathscr{H}^\phi(X)$, corresponding to X, is defined as $\mathscr{H}^\phi(X) = \bigcup_{o \in X^\circ} \phi_*(H_o) \times \phi_\circ(o)$. Note that the domain $\mathscr{H}$, for standard `NI`, is the instantiation of $\mathscr{H}^\phi$, where ϕ_* maps any set to $\top$ and $\phi_\circ$ maps any set to itself.

Let $\phi \in uco(\wp(\mathbb{I}))$, selecting the inputs on which to check non-interference, then we define the following completeness equation

$$\mathscr{H} \circ [\![P]\!] \circ \mathscr{H}^\phi = \mathscr{H} \circ [\![P]\!] \tag{8}$$

Now we can connect $\mathscr{H}^\phi$ to `DNI`:

Theorem 2.9 *P satisfies* `DNI` *w.r.t. ϕ iff $\mathscr{H} \circ [\![P]\!] \circ \mathscr{H}^\phi = \mathscr{H} \circ [\![P]\!]$.*

2.3 All together...

The following definition combines all the three dimensions obtaining the notion of `ANI` as a generalization of the standard one [17]. We can say that the idea of abstract non-interference is that a program P satisfies abstract non-interference relatively to a pair of observations η and ρ, and to a property ϕ, denoted $[\phi \vdash (\eta)P(\rho)]$, if, whenever the input values have the same property ϕ then the best correct approximation of the semantics of P, w.r.t. η in input and ρ in output, does not change. This captures precisely the intuition that ϕ-indistinguishable input values provide η,ρ-indistinguishable results, for this reason it can still be considered a non-interference policy.

Definition 2.10 ((Declassified) Abstract non-interference `DANI`)
Let $\phi, \eta \in uco(\wp(\mathbb{I})), \rho \in uco(\wp(\mathbb{O}_\circ))$.

$$\boxed{\begin{array}{c}\text{A program } P \text{ satisfies } [\phi \vdash (\eta)P(\rho)] \text{ w.r.t. } \pi_{\mathtt{I}} \text{ and } \pi_{\mathtt{O}} \text{ if} \\ \forall x_1, x_2 \in \mathbb{I}\,.\, \phi(x_1) = \phi(x_2) \;\Rightarrow\; \rho(([\![P]\!](\eta(x_1)))) = \rho(([\![P]\!](\eta(x_2))))\end{array}}$$

For instance, in Eq. 1 we have $\phi = \langle \mathbb{T}^*, id^\circ\rangle$, $\eta = id$ and $\rho = id$.In the following, we define closures on $\mathbb{V}^n$ by using closures on $\mathbb{V}$. In this case we abuse notation by supposing that $\rho(\langle x,y\rangle) = \langle \rho(x), \rho(y)\rangle$.

Example 2.11 *Consider the property Sign and Par defined in Sect. 1.Consider $\mathscr{I}_\circ = \{x\}$, $\mathscr{I}_* = \{y\}$ and $\mathbb{I} = \mathbb{Z}$. Let $\phi =$ Sign, $\eta =$ id, $\rho =$ Par, and consider the program fragment:*

$$P \overset{\mathit{def}}{=} x := 2 * x * y^2;$$

In the standard notion of non-interference there is a flow of information from variable y to variable x, since x depends on the value of y, i.e., the statement does not satisfy non-interference.
Let us consider $[\mathit{Sign} \vdash (\mathit{id})P(\mathit{Par})]$. If $\mathit{Sign}(\langle x,y\rangle) = \langle \mathit{Sign}(x), \mathit{Sign}(y)\rangle = \langle 0+, 0+\rangle$, then the possible outputs are always in `ev`*, indeed the result is always even because there is a multiplication by 2. The same holds if $\mathit{Sign}(\langle x,y\rangle) = \langle 0-, 0-\rangle$. Therefore any possible output value, with a fixed observable input, has the same observable abstraction in Par, which is* `ev`*. Hence $[\mathit{Sign} \vdash (\mathit{id})P(\mathit{Par})]$ holds.*

The completeness formalization. All the completeness characterizations provided embed the abstraction in a different position inside the completeness equation. This makes particularly easy to combine all the completeness characterizations in order to obtain the completeness formalization of DANI.

Theorem 2.12 *P satisfies* DANI *w.r.t.* $\phi, \eta \in uco(\wp(\mathbb{I}))$ *and* $\rho \in uco(\wp(\mathbb{O}))$ *iff*

$$\begin{array}{cc} \textit{Static attack}: & \textit{Dynamic attack}: \\ \mathscr{H} \circ [\![P]\!]^{\rho}_{\eta} \circ \mathscr{H}^{\phi} = \mathscr{H} \circ [\![P]\!]^{\rho}_{\eta} & \mathscr{H}_{\rho} \circ [\![P]\!] \circ \mathscr{H}^{\phi} = \mathscr{H}_{\rho} \circ [\![P]\!] \end{array}$$

3 Abstract Non-Interference in Language-based Security

In the previous sections we introduced non-interference as a generic notion that can be used/applied in many fields of computer science, i.e., wherever we have to analyze a dependency relation between two sets of variables. However, this notion was introduced in the field of language-based security [7]. In this context we consider the same partition of input and output data into two types: private/confidential (the set of values for variables of type H is denoted $\mathbb{V}^{\mathtt{H}}$) and public (the set of values for variables L is denoted $\mathbb{V}^{\mathtt{L}}$). In this case, non-interference requires that by observing the public output, a malicious attacker must not be able to disclose any information about the private input. It is straightforward to note that this is an immediate instantiation of the general notion we provided with attribute independent abstractions, where $\mathbb{I}_{\circ} = \mathbb{O}_{\circ} = \mathbb{V}^{\mathtt{L}}$ and $\mathbb{I}_{*} = \mathbb{O}_{*} = \mathbb{V}^{\mathtt{H}}$.

In the context of language-based security the limitation of the standard notion of non-interference is even more relevant, in particular we can observe that, in general, it results in a extremely restrictive policy. Indeed, non-interference policies require that any change upon confidential data must not be revealed through the observation of public data. There are at least two problems with this approach. On one side, many real systems are intended to leak some kind of information. On the other side, even if a system satisfies non-interference, static checking being approximate could reject it as insecure. Both of these observations address the problem of *weakening* the notion of non-interference for both characterizing the information that *is allowed* to flow, and modeling *weaker* attackers that can observe only some properties of public data. Abstract non-interference can provide formal tools for reasoning about both these kind of weakenings of non-interference in language-based security. There exists a large literature on the problem of weakening non-interference (see [24] and [25] for a formalization of the relation between abstract non-interference and existing approaches). In the following, we show how we can choose and combine the different policies obtaining the different notions of abstract non-interference introduced in the last years for language-based security.

3.1 *Who* is attacking?

Let us focus first on the attacker. By attacker we mean the agent aiming at learning some confidential information by *analyzing*, to the best of its possibilities, the system under attack. In general the attacker is characterized by the public observation of both input and output [15]. Depending on the dimension (observation or protection) we use for modeling the *input* attacker observation we obtain different notions of abstract non-interference. In particular, if the attacker *selects* the computations whose inputs have a fixed public property, we obtain the so called *narrow* non-interference [15]. If, instead the attacker performs a *static* analysis of the code with a generic input observation that may not be the same as the output observation, then we obtain the so called (strictly) abstract approach. In both cases the attacker is characterised by means of two abstractions: the abstract observation of the public *input* δ and the abstract observation of the public *output* ρ, both modelled as abstract domains.

3.1.1 The *narrow* approach to non-interference

In [15, 16] the notion of *narrow (abstract) non-interference* (`NANI` for short) represents a first weakening of standard non-interference relative to a given model of an attacker. The idea behind this notion is to consider attackers that can only dynamically analyze the program by analysing properties of the collected executions. In particular the input public observation *selects* the computations that have to satisfy non-interference, while the public output observation represents *dynamic observational* power of the attacker. Formally, given $\delta = \phi_{\mathtt{L}}$, $\rho \in uco(\wp(\mathbb{V}^{\mathtt{L}}))$, respectively, the input and the output observation, we say that a program P satisfies narrow non-interference (`NANI`), written $[\phi_{\mathtt{L}}]P(\rho)$, if

$$\boxed{\forall x_1, x_2 \in \mathbb{V} \,.\, \phi_{\mathtt{L}}(x_1^{\mathtt{L}}) = \phi_{\mathtt{L}}(x_2^{\mathtt{L}}) \;\Rightarrow\; \rho((\llbracket P \rrbracket(x_1))^{\mathtt{L}}) = \rho((\llbracket P \rrbracket(x_2))^{\mathtt{L}})}$$

This notion is a particular instantiation of `DANI` where $\eta = id$ and $\phi = \langle id_{\mathtt{H}}, \phi_{\mathtt{L}} \rangle$ (where $id_{\mathtt{H}} \overset{\text{def}}{=} \lambda x \in \mathbb{V}^{\mathtt{H}} \,.\, x$ and id is defined on $\mathbb{V}$). This notion corresponds to other weakenings of non-interference existing in the literature (e.g., PERs [32]).

3.1.2 The *abstract* approach to non-interference

A different abstract interpretation-based approach to non-interference can be obtained by modelling attackers as static analyzers of programs. In this case, we check non-interference by considering the best correct approximation of the program semantics in the abstract domains modelling the attacker. Formally, the idea is to compute the semantics on abstract values, obtaining again a notion of non-interference where only the private input can vary. What we obtain is a policy such that when the attacker is able to *observe* the property $\delta = \eta_{\mathtt{L}}$ of public input, and the property ρ of public output, then no information flow concerning the private input is detected by observing the public output. We call this notion *abstract non-interference* (`ANI` for short). A program P satisfies abstract non-interference, written $(\eta_{\mathtt{L}})P(\rho)$, if

$$\boxed{\forall x_1, x_2 \in \mathbb{V} \,.\, x_1^{\mathtt{L}} = x_2^{\mathtt{L}} \;\Rightarrow\; \rho(\llbracket P \rrbracket(x_1^{\mathtt{H}}, \eta_{\mathtt{L}}(x_1^{\mathtt{L}}))^{\mathtt{L}}) = \rho(\llbracket P \rrbracket(x_2^{\mathtt{H}}, \eta_{\mathtt{L}}(x_2^{\mathtt{L}}))^{\mathtt{L}})}$$

where we abuse notation denoting by $\llbracket P \rrbracket$ the additive lift, to sets of states, of the denotational semantics of P. This notion is an instantiation of `DANI` where $\phi = id$ and $\eta = \langle id_{\mathtt{H}}, \eta_{\mathtt{L}} \rangle$. Note that $(id)P(id)$ (equivalent to $[id]P(id)$) models exactly (standard) non-interference.

Proposition 3.1 *The notion of* `ANI` $(\delta)P(\rho)$ *defined above is equivalent to the standard notion of* `ANI` *[15], i.e.,* $\forall x_1, x_2 \in \mathbb{V} \,.\, \delta(x_1^{\mathtt{L}}) = \delta(x_2^{\mathtt{L}}) \;\Rightarrow\; \rho(\llbracket P \rrbracket(x_1^{\mathtt{H}}, \delta(x_1^{\mathtt{L}}))^{\mathtt{L}}) = \rho(\llbracket P \rrbracket(x_2^{\mathtt{H}}, \delta(x_2^{\mathtt{L}}))^{\mathtt{L}})$.

PROOF. We have to prove that $\forall x_1, x_2 \in \mathbb{V}$ we have that $(1) x_1^{\mathtt{L}} = x_2^{\mathtt{L}} \;\Rightarrow\; \rho(\llbracket P \rrbracket(x_1^{\mathtt{H}}, \delta(x_1^{\mathtt{L}}))^{\mathtt{L}}) = \rho(\llbracket P \rrbracket(x_2^{\mathtt{H}}, \delta(x_2^{\mathtt{L}}))^{\mathtt{L}})$ iff $(2) \delta(x_1^{\mathtt{L}}) = \delta(x_2^{\mathtt{L}}) \;\Rightarrow\; \rho(\llbracket P \rrbracket(x_1^{\mathtt{H}}, \delta(x_1^{\mathtt{L}}))^{\mathtt{L}}) = \rho(\llbracket P \rrbracket(x_2^{\mathtt{H}}, \delta(x_2^{\mathtt{L}}))^{\mathtt{L}})$. Suppose (1) holds, and consider $x_1^{\mathtt{L}} \neq x_2^{\mathtt{L}}$ such that $\delta(x_1^{\mathtt{L}}) = \delta(x_2^{\mathtt{L}})$. From (1) we have that $\rho(\llbracket P \rrbracket(x_1^{\mathtt{H}}, \delta(x_1^{\mathtt{L}}))^{\mathtt{L}}) = \rho(\llbracket P \rrbracket(x_2^{\mathtt{H}}, \delta(x_1^{\mathtt{L}}))^{\mathtt{L}})$, but since $\delta(x_1^{\mathtt{L}}) = \delta(x_2^{\mathtt{L}})$ we have also $\rho(\llbracket P \rrbracket(x_1^{\mathtt{H}}, \delta(x_1^{\mathtt{L}}))^{\mathtt{L}}) = \rho(\llbracket P \rrbracket(x_2^{\mathtt{H}}, \delta(x_2^{\mathtt{L}}))^{\mathtt{L}})$. Suppose (2) holds and that $x_1^{\mathtt{L}} = x_2^{\mathtt{L}}$, then trivially $\delta(x_1^{\mathtt{L}}) = \delta(x_2^{\mathtt{L}})$ and by (2) we have $\rho(\llbracket P \rrbracket(x_1^{\mathtt{H}}, \delta(x_1^{\mathtt{L}}))^{\mathtt{L}}) = \rho(\llbracket P \rrbracket(x_2^{\mathtt{H}}, \delta(x_2^{\mathtt{L}}))^{\mathtt{L}})$. □

3.2 *What* the attacker may disclose?

In this section, we focus on another important aspect: what is released or protected. Again, depending on which dimension (observation or protection) we use for modeling the information to protect we obtain different approaches to declassification [15, 25]. In particular, if the observation dimension is specified, then we are fixing the property that must not be released, namely we model the property whose variation must not be observable, obtaining the so called *block* approach to non-interference. On the other hand, if we use the protection dimension, then we are fixing the property that may be released, namely the property of private inputs such that non-interference has to be satisfied for all the private inputs with the same property. In other words, the property whose variation *may* interfere. In this case we obtain the so called *allow* approach to non-interference [15]. Note that in this context, we consider declassification while ignoring *where* [33] declassification takes place.

3.2.1 The *block* approach to declassification

Let us describe the *block approach* introduced in the original notion of abstract non-interference [15]. Note that in standard non-interference we have to protect the *value* of private data. If we interpret this fact from the point of view of what we have to keep secret, then we can say that we want to block the *identity* property of the private data domain. In the definition, we make the private input range in the domain of values and we check if these changes are detectable from the observation of the public output. Suppose, for instance, that we are interested in keeping secret the parity of input private data. Then we make the private input range over the abstract domain of parity, so we check if there is a variation in the public output only when the parity of the private input changes[8]. Hence, the fact that we want to protect parity is modelled by observing that the *distinction* between even and odd private inputs corresponds exactly to what must not be visible to a public output observer. Formally, consider an abstract domain $\eta_{\mathtt{H}} \in uco(\mathbb{V}^{\mathtt{H}})$ modelling the private information we want to keep secret. A program P satisfies non-interference declassified via blocking (B-DNI for short), written $(id)P(\eta_{\mathtt{H}} \rightsquigarrow\!\![\![id)$, if

$$\boxed{\forall x_1, x_2 \in \mathbb{V}\,.\, x_1^{\mathtt{L}} = x_2^{\mathtt{L}} \;\Rightarrow\; [\![P]\!](\eta_{\mathtt{H}}(x_1^{\mathtt{H}}), x_1^{\mathtt{L}})^{\mathtt{L}} = [\![P]\!](\eta_{\mathtt{H}}(x_2^{\mathtt{H}}), x_2^{\mathtt{L}})^{\mathtt{L}}}$$

This is a particular instantiation of DANI where $\phi = id$ and $\eta = \langle \eta_{\mathtt{H}}, id_{\mathtt{L}} \rangle$. We can obtain the narrow and the abstract declassified forms simply by combining this notion with each one of the previous notions [15]. In this case it is the semantics that *blocks* the flow of information, exactly as the square operation hides the input sign of an integer.

3.2.2 The *allow* approach to declassification

Finally, let us introduce the *allow approach* to declassification. This is a well-known approach, which has been introduced and enforced in several ways in the literature [24, 33]. The idea is to fix which aspects of the private information can be observed by an unclassified observer. From this point of view, the standard notion of non-interference, where nothing has to be observed, can be interpreted by saying that only the property $\mathbb{T}^*$ (i.e., "I don't know the private property") is declassified. This corresponds to

[8]This idea was partially introduced in the use of *abstract variables* in the semantic approach of Joshi and Leino [22], where the variables represent set of values instead of single values. But in [22] only an example of these ideas is provided.

saying that we have to check non-interference only for those private inputs that have the same declassified property, noting that all the values are mapped to $\top$ and hence have the same abstract property. Suppose, for instance, that we want to downgrade the parity. This means that we do not care if the observer sees any change due to the variation of this private input property. For this reason, we only check the variations of the output when the private inputs have the same parity property. Formally, consider an abstract domain $\phi_{\mathtt{H}}$ modelling the private information that is downgraded. A program P satisfies non-interference declassified via allowing (A-DNI for short), written $(id)P(\phi_{\mathtt{H}} \Rightarrow id)$, if

$$\boxed{\forall x_1, x_2 \in \mathbb{V}\,.\, x_1^{\mathtt{L}} = x_2^{\mathtt{L}} \;\wedge\; \phi_{\mathtt{H}}(x_1^{\mathtt{H}}) = \phi_{\mathtt{H}}(x_2^{\mathtt{H}}) \;\Rightarrow\; [\![P]\!](x_1^{\mathtt{H}}, x_1^{\mathtt{L}})^{\mathtt{L}} = [\![P]\!](x_2^{\mathtt{H}}, x_2^{\mathtt{L}})^{\mathtt{L}}}$$

Also this notion is an instantiation of DANI where $\eta = id$ and $\phi = \langle \phi_{\mathtt{H}}, id_{\mathtt{L}} \rangle$. This is a weakened form of standard NI where the test cases are reduced, allowing some confidential information to flow.

In order to make clear the differences and the analogies between the several notions introduced, we collect together, in the next table, all these notions.

OBSERVATION DIMENSION	DECLASSIFICATION DIMENSION	PROPERTY
Input: Dynamic protection of inputs ($\phi_{\mathtt{L}}$) *Output:* Dynamic analysis of computations (ρ)	No Declassification	NANI $[\phi_{\mathtt{L}}]P(\rho)$
Static analysis of code: $\eta_{\mathtt{L}}$ in input, ρ in output	No Declassification	ANI $(\eta_{\mathtt{L}})P(\rho)$
No abstraction	$\eta_{\mathtt{H}}$ to protect	B-DNI $(id)P(\eta_{\mathtt{H}} \rightsquigarrow [\![id)$
No abstraction	$\phi_{\mathtt{H}}$ declassified	A-DNI $(id)P(\phi_{\mathtt{H}} \Rightarrow id)$

3.3 *Where* the attacker may observe?

In this section, we show that we can exploit the semantic dimension in order to model also *where* the attacker may observe public data. In particular, we need to consider trace-based NI for fixing also the observable program points, where the attacker in some way can access the intermediate observable computations. Let us recall trace-based NI for language-based security

$$\forall j \in \mathtt{O}.\, \forall s_1, s_2 \in \Sigma_{\vdash}\,.\, s_1^{\mathtt{L}} = s_2^{\mathtt{L}} \;\Rightarrow\; post_P^j(s_1)^{\mathtt{L}} = post_P^j(s_2)^{\mathtt{L}}$$

and its completeness characterization in terms of weakest precondition semantics:

$$\mathscr{H} \circ \widetilde{pre}^j \circ \mathscr{H} = \widetilde{pre}^j \circ \mathscr{H}.$$

These results say that, in order to check DNI on traces, we would have to make an analysis for each observable program point j, combining, afterwards, the information disclosed. In order to make only one iteration on the program even when dealing with traces, our basic idea is to combine the weakest precondition semantics, computed at each observable point of the execution, together with the observation of public data made at the particular observation point.

We will describe our approach on our running example, Ex. 3.2, where o_i and p_i denote respectively the observation and the protection points of the program P.

Example 3.2

$$P = \left[\begin{array}{lll} p_0 \rightarrow & \quad\{h_2 \text{ mod } 2 = a\} & \leftarrow o_0 \\ & h_1 := h_2; & \\ & h_2 := h_2 \text{ mod } 2; & \\ & l_1 := h_2; & \\ & h_2 := h_1 & \\ & l_2 := h_2; & \\ & l_2 := l_1 & \\ & \quad\{l_1 = l_2 = a\} & \leftarrow o_6 \end{array}\right.$$

In this example, the observation in output of l_1 or l_2 allows us to derive the information about the parity of the secret input h_2.

In the presence of explicit declassifications, it allows a precise characterization of the relation between the *where* and the *what* dimensions of non-interference policies. The idea is to track (by using the wlp computation) the information disclosed in each observation point till the beginning of the computation, and then to compare it with the corresponding declassification policy. The next example shows how we track the information disclosed in each observable program point. We consider as the set of observable program points, O, the same set used for gradual release, namely, the program points corresponding to low events. However, in general, our technique allows O to be any set of program points. When there is more than one observation point, we will use the notation $[\Phi]^O$ to denote that the information described by the assertion Φ can be derived from the set of observation points in O.

Example 3.3

$$P = \left[\begin{array}{lll} p_0 \rightarrow & \quad\{[h_2 = b]^{o_5}, [h_2 \text{ mod } 2 = a]^{o_3,o_5,o_6}, [l_2 = c]^{o_3,o_0}, [l_1 = d]^{o_0}\} & \leftarrow o_0 \\ & h_1 := h_2; & \\ & \quad\{[h_1 = b]^{o_5}, [h_2 \text{ mod } 2 = a]^{o_3,o_5,o_6}, [l_2 = c]^{o_3}\} & \\ & h_2 := h_2 \text{ mod } 2; & \\ & \quad\{[h_1 = b]^{o_5}, [h_2 = a]^{o_3,o_5,o_6}, [l_2 = c]^{o_3}\} & \\ & l_1 := h_2; & \\ & \quad\{[h_1 = b]^{o_5}, [l_1 = a]^{o_3,o_5,o_6}, [l_2 = c]^{o_3}\} & \leftarrow o_3 \\ & h_2 := h_1 & \\ & \quad\{[h_2 = b]^{o_5}, [l_1 = a]^{o_5,o_6}\} & \\ & l_2 := h_2; & \\ & \quad\{[l_1 = a]^{o_5,o_6}, [l_2 = b]^{o_5}\} & \leftarrow o_5 \\ & l_2 := l_1 & \\ & \quad\{l_1 = l_2 = a\} & \leftarrow o_6 \end{array}\right.$$

For instance, at observation point o_5, $[l_1 = a]^{o_5,o_6}$ is obtained either via wlp calculation of $l_2 := l_1$ from o_6, or via the direct observation of public data in o_5, while $[l_2 = b]^{o_5}$ — where b is an arbitrary symbolic value — is only due to the observation of the value l_2 in o_5. The assertion in o_3 — $[h_1 = b]^{o_5}, [l_1 = a]^{o_3,o_5,o_6}$ — is obtained by computing the wlp semantics $Wlp(h_2 := h_1, ([h_2 = b]^{o_5}, [l_1 = a]^{o_3,o_5,o_6}))$, while $[l_2 = c]^{o_3}$ is the observation in o_3. Similarly, we can derive all the other assertions.

It is worth noting, that the attacker is more powerful than the one considered in Example 3.2. In fact, in this case, the possibility of observing l_2 in the program point o_5 allows to derive the exact (symbolic) value of h_2 ($h_2 = b$ where b is the value observed in o_5). This was not possible in Example 3.2 based on simple input-output, since the value of l_2 was lost in the last assignment.

Now, we can use the information disclosed in each observation point, characterized by computing the wlp semantics, in order to check if the corresponding declassification policy is satisfied or not. This is obtained simply by comparing the abstraction modelling the declassification with the state abstraction corresponding to the information released in the lattice of abstract interpretations. In the sequel, we will consider a slight extension of a standard imperative language with explicit syntax for declassification as in Jif [29] or in the work on delimited release [30]. Such syntax often takes the form $l := \text{declassify}(e(h))$, where l is a public variable, h is a secret variable and e is an expression containing the variable h. This syntax means that the final value of $e(h)$, corresponding to some information about h, can safely made public assigning it to a public variable l.

Example 3.4 *Consider the program in Example 3.3, with the only difference that in o_3 the statement $l_1 := h_2$ is substituted with $l_1 := declassify(h_2)$. In this case, the corresponding declassification policy is $\phi_3 = id_{h_2}$. Now we can compare this policy with the private information disclosed in o_3, which is $h_2 \bmod 2 = a$ (h_2's parity) and therefore we conclude that the declassification policy in o_3 is satisfied because parity is more abstract than identity. Nevertheless, the program releases the information $h_2 = b$ in the program point o_5. This means that the security of the programs depends on the kind of localized declassification we consider. For incremental declassification the program is secure since the release is licensed by the previous declassification since the observation point o_5, where we release information corresponds to the declassification policy $\phi_5 = \phi_3 = id_{h_2}$, while for localized declassification the program is insecure since there isn't a corresponding declassification policy at o_5 that licenses the release.*

3.4 Certifying abstract non-interference

In this section we briefly recall the main techniques proposed for certifying different aspects of ANI in language-based security.

Characterizing attackers. In [15], a method for deriving the most concrete output observation for a program, given the input one, is provided. In particular, we have interference when the attacker observes too much, namely when it can *distinguish* elements due to different confidential inputs. The idea is to collect together all such observations. We don't mean here to enter in the details of this characterization (see [15] and [16]), we simply describe the different steps we perform in order to characterize the most concrete harmless output (dynamic) observer. By dynamic we mean that we can only characterize the output observation ρ but we cannot characterize the input one, η, which remain fixed.

1. First of all we define the sets of elements that, depending on the fixed (abstract) input, must not be distinguished. In other words, if the inputs have to agree simply on the public part, then we collect in the same set all the outputs due to the variation of the confidential inputs. For instance, for the property $\texttt{ANI}\ (\delta)P(\rho)$ we are interested in the elements of

$$\Upsilon^{P}_{\texttt{ANI}(\delta)} \stackrel{\text{def}}{=} \left\{ \left\{ [\![P]\!](h,\delta(l)) \,\middle|\, h \in \mathbb{V}^{\texttt{H}} \right\} \,\middle|\, l \in \mathbb{V}^{\texttt{L}} \right\}$$

Namely, each $Y \in \Upsilon^P_{\texttt{ANI}(\delta)}$ is a set of elements indistinguishable for guaranteeing non-interference.

2. Let Π the property to analyze (`ANI` for instance), the maximal harmless observation is the set of all the elements satisfying the predicate $Secr^{\Pi}_P$:

$$\forall X \in \wp(\mathbb{V}^{\mathtt{L}}) \,.\, Secr^{\Pi}_P(X) \;\Leftrightarrow\; (\exists Z \in \Upsilon^P_{\Pi}.Z \subseteq X \;\Rightarrow\; \forall W \in \Upsilon^P_{\Pi}.\, W \subseteq X)$$

Namely the maximal subset of the power set of program states such that any of its elements can only completely contain sets in Υ^P_{Π}, namely does not break indistinguishable sets.

Characterizing information released. Abstract non-interference can be exploited also for characterizing the maximal amount of confidential information released, modelled in terms of the most concrete confidential property that can be analyzed by a given attacker [15]. The idea is to find the maximum amount of information disclosed by computing the most abstract property on confidential data which has to be declassified in order to guarantee secrecy. In other words, we characterize the minimal aggregations of confidential inputs that do not generate any variation in the public output.

1. We collect together all and only the confidential values that do not generate a variation in the output observation, for instance for `ANI` we consider

$$\Pi_{\mathtt{P}}(\delta,\rho) \stackrel{\text{def}}{=} \{\langle\{h \in \mathbb{V}^{\mathtt{H}} \mid \rho([\![P]\!](\langle h, \delta(l)\rangle)^{\mathtt{L}}) = A\}, \delta(l)\rangle \mid l \in \mathbb{V}^{\mathtt{L}},\; A \in \rho\}$$

which, for each public input l and for each output observation A gathers all the confidential inputs generating precisely the observation A.

2. We use this information for generating the abstraction Φ, of confidential inputs such that any pair of inputs in the same set does not generate a flow, while any pair of inputs belonging to different sets do generate a flow:

$$\Pi_{\mathtt{P}}(\delta,\rho)_{|L} \stackrel{\text{def}}{=} \{H \mid \langle H, L\rangle \in \Pi_{\mathtt{P}}(\delta,\rho)\} \qquad \Phi \stackrel{\text{def}}{=} \mathscr{P}(\textstyle\bigsqcap_{L\in\delta} \mathscr{M}(\Pi_{\mathtt{P}}(\delta,\rho)_{|L}))$$

This domain is the most abstract property that contains all the possible variations of confidential inputs that generate insecure information flows, and it is the most concrete such that each variation generates a flow. It uniquely represents the confidential information that flows into the public output [15].

A proof system for Abstract non-interference In the previous sections, we start from the program and in some way we characterize the abstract non-interference policy that can be satisfied by adding some weakenings, in the attacker model or in the information allowed to flow. In the literature, we can find another certification approach to abstract non-interference. In particular, in [17] the authors provide a *proof system* for certifying `ANI` in programming languages. In this way, they can prove, inductively on the syntactic structure of programs, whether a specific `ANI` policy is or not satisfied.

4 A promising approach in other security fields

In this section, we provide the informal intuition of how interference properties pervade two challenging IT security fields and therefore of how approaches based on abstract non-interference may be promising into these fields and deserve further research.

```
var cookie = document.cookie; /*initialisation of the cookie by the server*/
var dut;
if (dut == undefined) {dut = "";}
while(i<cookie.length) {
    switch(cookie[i]) {
       case 'a': dut += 'a'; break;
       case 'b': dut += 'b'; break;
       ...
  }
}      /* dut contains now copy of cookie*/
document.images[0].src =  "http://badsite/cookie?" + dut;
/* When the user click on the image dut is sent to the web server under the attackers control */
```

Figure 5: Code creating a XSS vulnerability.

Code injection. Among the major security threats in the web application context we have *code injection attacks*. Typically, these attacks inject some inputs, interpreted as executable code, which is not adequately checked and which *interfere* with the surrounding code in order to disclose information or to take the control of the system. Examples of these kinds of attacks are: SQL injections, Cross-site scripting (XSS), Command injections, etc. In order to protect a program from these kind of attacks we first can verify whether the code coming from potentially untrusted sources may interfere with the sensible components of the system. The idea of tainted analysis is that of following the potentially untrusted inputs (tainted) checking whether they get in contact with sensible containers or data. Instead, the idea based on (abstract) non-interference [4] uses the weakest precondition approach due to completeness characterization of non-interference [16, 25] introduced in Sect. 3.3 in order to characterize which variables, i.e., information, the injected code may manipulate in order to guarantee the absence of insecure information flows. The idea in this case is to take into account also the implicit protection that the code itself may provide. In [4] this idea is precisely formalized and here we describe it by using a XSS attack example. Suppose a user visits an untrusted web site in order to download a picture, where an attacker has inserted his own malicious Javascript code (Fig. 5), and execute it on the clients browser [34]. In the following we describe a simplified version. The Javascript code snippet in Fig. 5 can be used by the attacker to send cookies[9] of the user to a web server under the attackers control. By performing the analysis proposed in [4] we can show that confidentiality is violated since there is a (implicit) flow of information from private variable *cookie* towards the public variable *dut*. This corresponds to the sensitive information disclosed when *dut* is initialised to the empty string. We can also formally show that an attacker can exploit *dut* for disclosing other user confidential information. Suppose, for instance, the attacker to be interested in the *history* object together with its attributes[10].

[9]A cookie is a text string stored by a user's web browser. A cookie consists of one or more name-value pairs containing bits of information, sent as an HTTP header by a web server to a web browser (client) and then sent back unchanged by the browser each time it accesses that server. It can be used, for example, for authentication.

[10]The history object allows to navigate through the history of websites that a browser has visited.

An attack could loop over the elements of the *history* object and pass through variable *dut* all the web pages the client has had access to. Consider for example the injection code on the right.

```
<script language="JavaScript">
var dut = "";
for (i=0; i<history.length; i++){
    dut = dut + history.previous;
}   </script>
```

Hence the ANI-based analysis detect a code vulnerability that the attacker can exploit by inserting the code above just before the malicious code (Fig. 5) in the untrusted web page, getting both *history* and *cookie* through the variable *dut*. It is worth noting that this approach provides a theoretical model for the existing techniques used in practice for protecting code from XSS attacks [34].

Code obfuscation. Code obfuscation is increasing its relevance in security, providing an effective way for facing the problem of both source code protection and binary protection. Code obfuscation [8] aims at transforming programs in order to make them more difficult to analyze while preserving their functionality.
In [14], the authors introduce a generalized notion of obfuscation which, while preserving denotational semantics, obfuscates, namely transforms, an abstract property of interest, the one to protect (this is a further generalization of the obfuscation defined in [12]). Let P be a program, $[\![P]\!]$ its denotational semantics, and $[\![P]\!]^\rho$ the abstract semantics to obfuscate, namely ρ is the abstract semantic property to protect, then $\hat{t}$ is an obfuscator w.r.t. ρ if

$$[\![P]\!] = [\![\hat{t}(P)]\!] \qquad \text{and} \qquad [\![P]\!]^\rho \neq [\![\hat{t}(P)]\!]^\rho$$

Based on this notion of obfuscation, in [14] the authors propose to exploit the analysis of code fragments dependencies in order to obfuscate precisely the code *relevant* in the computation of the information to protect. This idea is based on the awareness that the effect of replacing code in programs can only be understood by the notion of dependency [1]. Hence, the idea is to model the effect of substituting program fragments with *equivalent* obfuscated structures by indirectly considering the ANI framework. This is an indirect application since it goes through the dependency analysis instead of the non-interference analysis, namely it is more related with slicing, which can be weakened in the so called *abstract slicing* [27, 26], strongly related to ANI[11].
The idea is to characterize the *dependency* between the computation of the values of the different variables x_i and the observable semantics of the program P. For this reason the notion of *stability* is defined [14], which specifies that a program P is stable w.r.t. the variables x_i, when any change of their abstract (computed) values does not change the observable semantics of the program.

The notion of stability corresponds to ANI, considering x_i as internal, and observing the whole output. In code obfuscation, we are interested in the negation of this property which is *instability*, i.e., there exist abstract values for x_i that cause a variation in the output observation. In other words, a variation of a fixed internal property η_* induces a variation in the observable output. This means that the code computing the variables x_i inducing instability is the portion of code that it is sufficient to obfuscate in order to modify the observation of the whole program, deceiving any observer. This is a sufficient condition since, due to instability, we guarantee that the output observation changes. Again this is a new approach to code obfuscation that surely deserves further research.

[11] Program slicing [35] is a program manipulation technique which extracts, from programs, statements relevant to a particular computation.

5 Conclusions

In this paper we provide a survey on the framework of abstract non-interference developed in the last years. This generalization of language-based non-interference was introduced in 2004 [15] as a semantic-based weakening of non-interference allowing some information to flow in order to accept programs, such as password checking, that even if considered acceptable where rejected w.r.t. standard non-interference. We have also shown that this new approach to non-interference was indeed strongly related with the main existing approaches to non-interference [24]. Since then, we realized that abstract non-interference was a powerful theoretical framework where it was possible to model and understand several aspects of non-interference policies, modeling also declassification and formally relating different non-interference policies, usually independently introduced [25]. This is proved also by the fact that our approach was probably of inspiration in different recent works such as [6], where declassification policies are modeled in a weakest precondition style, as we've done in [5], or in [3] where the authors model declassification as a weakening of what the attacker can observe of the input data.
Concluding, we believe that abstract non-interference is a really general framework that can provide useful theoretical bases for understanding "interactions" in different fields of security, as briefly shown in the last section of this paper. For instance, a challenge for the abstract non-interference approach is to extend it in order to cope with other security policies, such as those proposed by McLean [28] and or by Mantel [23], which would allow to consider interactions between functions instead of between variables. This kind of properties could be interesting, for instance, in malware detection for analyzing the interference between a malware and the execution environment [13].
Finally, we are obviously aware also of existing limitations of this approach. Among all, our approach is still quite far from a practical exploitation: it is powerful for understanding interference phenomena, but the cost of this power is the distance from a real implementation which is our next big challenge to face.

Acknowledgements. I would like to thank Dave Schmidt for the time he dedicated to me during my postdoc in Manhattan and for being so inspiring for the research I developed since then.
This work was partially supported by the PRIN 2010-2011 project ”Security Horizons”

References

[1] M. Abadi, A. Banerjee, N. Heintze & J. Riecke (1999): *A core calculus of dependency*. In: *Proc. of the 26th Annual ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages (POPL '99)*, ACM-Press, pp. 147–160. doi:10.1145/292540.292555

[2] A. Askarov & A. Sabelfeld (2007): *Gradual Release: Unifying Declassification, Encryption and Key Release Policies*. In: *Proc. IEEE Symp. on Security and Privacy*, IEEE Comp. Soc. Press, pp. 207–221. doi:10.1109/SP.2007.22

[3] M. Balliu, M. Dam & G. Le Guernic (2011): *Epistemic Temporal Logic for Information Flow Security*. In: *Proc. of the 2011 workshop on Programming languages and analysis for security*, ACM Press. doi:10.1145/2166956.2166962

[4] M. Balliu & I. Mastroeni (2010): *A Weakest Precondition Approach to Robustness*. *LNCS Transactions on Computational Science* 10, pp. 261 – 297. doi:10.1007/978-3-642-17499-5_11

[5] A. Banerjee, R. Giacobazzi & I. Mastroeni (2007): *What you lose is what you leak: Information leakage in declassifivation policies*. In: *Proc. of the 23th Internat. Symp. on Mathematical Foundations of Programming Semantics* (*MFPS '07*), *Electronic Notes in Theoretical Computer Science* 1514, Elsevier. doi:10.1016/j.entcs.2007.02.027

[6] A. Banerjee, D. A. Naumann & S. Rosenberg (2008): *Expressive Declassification Policies and Modular Static Enforcement*. In: *IEEE Symp. on Security and Privacy*, pp. 339 – 353. doi:10.1109/SP.2008.20

[7] E. S. Cohen (1977): *Information transmission in computational systems*. *ACM SIGOPS Operating System Review* 11(5), pp. 133–139. doi:10.1145/1067625.806556

[8] C. Collberg, C. D. Thomborson & D. Low (1998): *Manufactoring Cheap, Resilient, and Stealthy Opaque Constructs*. In: *Proc. of Conf. Record of the 25st ACM Symp. on Principles of Programming Languages* (*POPL '98*), ACM Press, pp. 184–196. doi:10.1145/268946.268962

[9] P. Cousot (2002): *Constructive design of a hierarchy of semantics of a transition system by abstract interpretation*. *Theor. Comput. Sci.* 277(1-2), pp. 47–103. doi:10.1016/S0304-3975(00)00313-3

[10] P. Cousot & R. Cousot (1977): *Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints*. In: *Conference Record of the 4th ACM Symposium on Principles of Programming Languages* (*POPL '77*), ACM Press, pp. 238–252. doi:10.1145/512950.512973

[11] P. Cousot & R. Cousot (1979): *Systematic design of program analysis frameworks*. In: *Conference Record of the 6th ACM Symposium on Principles of Programming Languages* (*POPL '79*), ACM Press, pp. 269–282. doi:10.1145/567752.567778

[12] M. Dalla Preda & R. Giacobazzi (2009): *Semantic-based Code Obfuscation by Abstract Interpretation*. *Journal of Computer Security* 17(6), pp. 855–908. doi:10.1007/11523468_107

[13] M. Dalla Preda & I. Mastroeni (2013): *Chasing Infections by Unveiling Program Dependencies*. In: *1st International Workshop on Interference and Dependence (ID '13)*.

[14] R. Giacobazzi, N. D. Jones & I. Mastroeni (2012): *Obfuscation by Partial Evaluation of Distorted Interpreters*. In O. Kiselyov & S. Thompson, editors: *Proc. of the ACM SIGPLAN Symp. on Partial Evaluation and Semantics-Based Program Manipulation (PEPM'12)*, ACM Press, pp. 63 – 72. doi:10.1145/2103746.2103761

[15] R. Giacobazzi & I. Mastroeni (2004): *Abstract Non-Interference: Parameterizing Non-Interference by Abstract Interpretation*. In: *Proc. of the 31st Annual ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages (POPL '04)*, ACM-Press, pp. 186–197. doi:10.1145/964001.964017

[16] R. Giacobazzi & I. Mastroeni (2010): *Adjoining classified and unclassified information by Abstract Interpretation*. *Journal of Computer Security* 18(5), pp. 751 – 797. doi:10.3233/JCS-2009-0382

[17] R. Giacobazzi & I. Mastroeni (2010): *A Proof System for Abstract Non-Interference*. *Journal of Logic and Computation* 20, pp. 449 – 479. doi:10.1093/logcom/exp053

[18] R. Giacobazzi & E. Quintarelli (2001): *Incompleteness, counterexamples and refinements in abstract model-checking*. In P. Cousot, editor: *Proc. of The 8th Internat. Static Analysis Symp. (SAS'01)*, *Lecture Notes in Computer Science* 2126, Springer-Verlag, pp. 356–373. doi:10.1007/3-540-47764-0_20

[19] R. Giacobazzi, F. Ranzato & F. Scozzari. (2000): *Making Abstract Interpretation Complete*. *Journal of the ACM* 47(2), pp. 361–416. doi:10.1145/333979.333989

[20] J. A. Goguen & J. Meseguer (1982): *Security policies and security models*. In: *Proc. IEEE Symp. on Security and Privacy*, IEEE Comp. Soc. Press, pp. 11–20.

[21] S. Hunt & I. Mastroeni (2005): *The PER model of Abstract Non-Interference*. In C. Hankin & I. Siveroni, editors: *Proc. of The 12th Internat. Static Analysis Symp. (SAS '05)*, *Lecture Notes in Computer Science* 3672, Springer-Verlag, pp. 171–185. doi:10.1007/11547662_13

[22] R. Joshi & K. R. M. Leino (2000): *A semantic approach to secure information flow*. *Science of Computer Programming* 37, pp. 113–138. doi:10.1016/S0167-6423(99)00024-6

[23] H. Mantel (2000): *Possibilistic definitions of security – an assemply kit –*. In: *Proc. of the IEEE Computer Security Foundations Workshop*, IEEE Comp. Soc. Press, pp. 185–199. doi:10.1109/CSFW.2000.856936

[24] I. Mastroeni (2005): *On the Rôle of Abstract Non-interference in Language-Based Security*. In K. Yi, editor: *Third Asian Symp. on Programming Languages and Systems (APLAS '05)*, *Lecture Notes in Computer Science* 3780, Springer-Verlag, pp. 418–433. doi:10.1007/11575467_27

[25] I. Mastroeni & A. Banerjee (2011): *Modelling Declassification Policies using Abstract Domain Completeness*. *Mathematical Structures in Computer Science* 21(6), pp. 1253 – 1299. doi:10.1017/S096012951100020X

[26] I. Mastroeni & D. Nikolic (2010): *An Abstract Unified Framework for (Abstract) Program Slicing*. In: *12th International Conference on Formal Engineering Methods, ICFEM 201*, *Lecture Notes in Computer Science* 6447, Spinger-Verlag, pp. 452–467. doi:10.1007/978-3-642-16901-4_30

[27] I. Mastroeni & D. Zanardini (2008): *Data dependencies and program slicing: From syntax to abstract semantics*. In: *Proc. of the ACM SIGPLAN Symp. on Partial Evaluation and Semantics-Based Program Manipulation (PEPM'08)*, ACM Press, pp. 125 – 134. doi:10.1145/1328408.1328428

[28] J. McLean (1996): *A general theory of composition for a class of "possibilistic" properties*. *IEEE Transactions on Software Engineering* 22(1), pp. 53 – 67. doi:10.1109/32.481534

[29] A. C. Myers, S. Chong, N. Nystrom, L. Zheng & S. Zdancewic: *Jif: Java information flow. Software release*. Available at `http://www.cs.cornell.edu/jif`.

[30] A. Sabelfeld & A. C. Myers (2004): *A model for delimited information release*. In N. Yonezaki K. Futatsugi, F. Mizoguchi, editor: *Proc. of the International Symp. on Software Security (ISSS'03)*, *Lecture Notes in Computer Science* 3233, Springer-Verlag, pp. 174–191. doi:10.1007/978-3-540-37621-7_9

[31] A. Sabelfeld & A.C. Myers (2003): *Language-based information-flow security*. *IEEE J. on selected ares in communications* 21(1), pp. 5–19. doi:10.1109/JSAC.2002.806121

[32] A. Sabelfeld & D. Sands (2001): *A PER Model of Secure Information Flow in Sequential Programs*. *Higher-Order and Symbolic Computation* 14(1), pp. 59–91. doi:10.1023/A:1011553200337

[33] A. Sabelfeld & D. Sands (2007): *Declassification: Dimensions and Principles*. *J. of Computer Security*. doi:10.3233/JCS-2009-0352

[34] P. Vogt, F. Nentwich, N. Jovanovic, E. Kirda, C. Krügel & G. Vigna (2007): *Cross Site Scripting Prevention with Dynamic Data Tainting and Static Analysis*. In: *NDSS*.

[35] M. Weiser (1981): *Program slicing*. In: *ICSE '81: Proceedings of the 5th international conference on Software engineering*, IEEE Press, pp. 439–449.